16 Using the platform features, part 2

This chapter covers

- Using cookies to store data that will be presented in subsequent requests
- Using sessions to identify related requests and store associated data
- Working with HTTPS requests
- Limiting the rate of requests processed by endpoints
- Responding to exceptions and errors
- Filtering requests based on the host header

In this chapter, I continue to describe the basic features provided by the ASP.NET Core platform. I explain how cookies are used and how the user's consent for tracking cookies is managed. I describe how sessions provide a robust alternative to basic cookies, how to use and enforce HTTPS requests, how to deal with errors, and how to filter requests based on the `Host` header. Table 16.1 provides a guide to the chapter.

Table 16.1 Chapter guide

Problem	Solution	Listing
Using cookies	Use the context objects to read and write cookies.	1–3
Managing cookie consent	Use the consent middleware.	4–6
Storing data across requests	Use sessions.	7, 8
Securing HTTP requests	Use the HTTPS middleware.	9–13
Restrict the number of requests handled by the application	Use the rate limiting middleware	14
Handling errors	Use the error and status code middleware.	15–20
Restricting a request with the host header	Set the `AllowedHosts` configuration setting.	21

16.1 Preparing for this chapter

In this chapter, I continue to use the `Platform` project from chapter 15. To prepare for this chapter, replace the contents of the `Program.cs` file with the contents of listing 16.1, which removes the middleware and services from the previous chapter.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/manningbooks/proasp.net-core-7. See chapter 1 for how to get help if you have problems running the examples.

Listing 16.1 Replacing the contents of the Program.cs file in the Platform folder

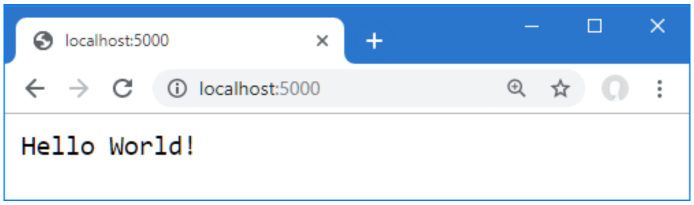
```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapFallback(async context =>
    await context.Response.WriteAsync("Hello World!"));

app.Run();
```

Start the application by opening a new PowerShell command prompt, navigating to the folder that contains the `Platform.csproj` file, and running the command shown in listing 16.2.

Listing 16.2 Starting the ASP.NET Core runtime

```
dotnet run
```

Open a new browser window and use it to request http://localhost:5000, which will produce the response shown in figure 16.1.


localhost:5000

Hello World!

Figure 16.1 Running the example application

16.2 Using cookies

Cookies are small amounts of text added to responses that the browser includes in subsequent requests. Cookies are important for web applications because they allow features to be developed that span a series of HTTP requests, each of which can be identified by the cookies that the browser sends to the server.

ASP.NET Core provides support for working with cookies through the `HttpRequest` and `HttpResponse` objects that are provided to middleware components. To demonstrate, listing 16.3 changes the routing configuration in the example application to add endpoints that implement a counter.

Listing 16.3 Using cookies in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/cookie", async context => {
    int counter1 =
        int.Parse(context.Request.Cookies["counter1"] ?? "0") + 1;
    context.Response.Cookies.Append("counter1", counter1.ToString(),
        new CookieOptions {
            MaxAge = TimeSpan.FromMinutes(30)
        });
    int counter2 =
        int.Parse(context.Request.Cookies["counter2"] ?? "0") + 1;
    context.Response.Cookies.Append("counter2", counter2.ToString(),
        new CookieOptions {
            MaxAge = TimeSpan.FromMinutes(30)
        });
    await context.Response
        .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
});

app.MapGet("clear", context => {
    context.Response.Cookies.Delete("counter1");
    context.Response.Cookies.Delete("counter2");
    context.Response.Redirect("/");
    return Task.CompletedTask;
});

app.MapFallback(async context =>
    await context.Response.WriteAsync("Hello World!"));

app.Run();
```

The new endpoints rely on cookies called `counter1` and `counter2`. When the `/cookie` URL is requested, the middleware looks for the cookies and parses the values to an `int`. If there is no cookie, a fallback zero is used.

```
...
int counter1 = int.Parse(context.Request.Cookies["counter1"] ?? "0") + 1;
...
```

Cookies are accessed through the `HttpRequest.Cookies` property, where the name of the cookie is used as the key. The value retrieved from the cookie is incremented and used to set a cookie in the response, like this:

```
...
context.Response.Cookies.Append("counter1", counter1.ToString(),
    new CookieOptions {
        MaxAge = TimeSpan.FromMinutes(30)
});
...
```

Cookies are set through the `HttpResponse.Cookies` property and the `Append` method creates or replaces a cookie in the response. The arguments to the `Append` method are the name of the cookie, its value, and a `CookieOptions` object, which is used to configure the cookie. The `CookieOptions` class defines the properties described in table 16.2, each of which corresponds to a cookie field.

NOTE Cookies are sent in the response header, which means that cookies can be set only before the response body is written, after which any changes to the cookies are ignored.

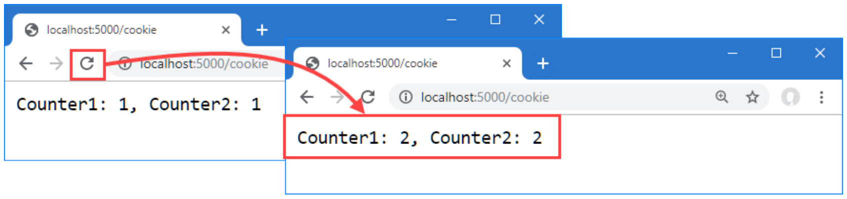
Table 16.2 The CookieOptions properties

Name	Description
`Domain`	This property specifies the hosts to which the browser will send the cookie. By default, the cookie will be sent only to the host that created the cookie.
`Expires`	This property sets the expiry for the cookie.
`HttpOnly`	When `true`, this property tells the browser not to include the `cookie` in requests made by JavaScript code.
`IsEssential`	This property is used to indicate that a cookie is essential, as described in the "Managing Cookie Consent" section.
`MaxAge`	This property specifies the number of seconds until the cookie expires. Older browsers do not support cookies with this setting.
`Path`	This property is used to set a URL path that must be present in the request before the cookie will be sent by the browser.
`SameSite`	This property is used to specify whether the cookie should be included in cross-site requests. The values are `Lax`, `Strict`, and `None` (which is the default value).
`Secure`	When `true`, this property tells the browser to send the cookie using HTTPS only.

The only cookie option set in listing 16.3 is `MaxAge`, which tells the browser that the cookies expire after 30 minutes. The middleware in listing 16.3 deletes the cookies when the `/clear` URL is requested, which is done using the `HttpResponse.Cookie.Delete` method, after which the browser is redirected to the / URL.

```
...
app.MapGet("clear", context => {
    context.Response.Cookies.Delete("counter1");
    context.Response.Cookies.Delete("counter2");
    context.Response.Redirect("/");
    return Task.CompletedTask;
});
...
```

Restart ASP.NET Core and navigate to http://localhost:5000/cookie. The response will contain cookies that are included in subsequent requests, and the counters will be incremented each time the browser is reloaded, as shown in figure 16.2. A request for http://localhost:5000/clear will delete the cookies, and the counters will be reset.



```
Counter1: 1, Counter2: 1
```

```
Counter1: 2, Counter2: 2
```

Figure 16.2 Using a cookie

16.2.1 Enabling cookie consent checking

The EU General Data Protection Regulation (GDPR) requires the user's consent before nonessential cookies can be used. ASP.NET Core provides support for obtaining consent and preventing nonessential cookies from being sent to the browser when consent has not been granted. The options pattern is used to create a policy for cookies, which is applied by a middleware component, as shown in listing 16.4.

> **CAUTION** Cookie consent is only one part of GDPR. See https://en.wikipedia.org/wiki/General_Data_Protection_Regulation for a good overview of the regulations.

Listing 16.4 Enabling cookie consent in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<CookiePolicyOptions>(opts => {
    opts.CheckConsentNeeded = context => true;
});

var app = builder.Build();

app.UseCookiePolicy();
app.MapGet("/cookie", async context => {
    int counter1 =
        int.Parse(context.Request.Cookies["counter1"] ?? "0") + 1;
    context.Response.Cookies.Append("counter1", counter1.ToString(),
        new CookieOptions {
            MaxAge = TimeSpan.FromMinutes(30),
            IsEssential = true
        });
    int counter2 =
        int.Parse(context.Request.Cookies["counter2"] ?? "0") + 1;
    context.Response.Cookies.Append("counter2", counter2.ToString(),
        new CookieOptions {
            MaxAge = TimeSpan.FromMinutes(30)
        });
    await context.Response
        .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
});

app.MapGet("clear", context => {
    context.Response.Cookies.Delete("counter1");
    context.Response.Cookies.Delete("counter2");
    context.Response.Redirect("/");
    return Task.CompletedTask;
});

app.MapFallback(async context =>
    await context.Response.WriteAsync("Hello World!"));

app.Run();
```

The options pattern is used to configure a `CookiePolicyOptions` object, which sets the overall policy for cookies in the application using the properties described in table 16.3.

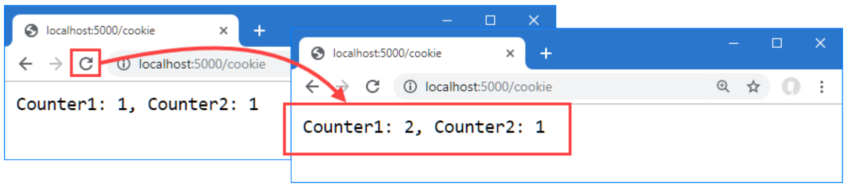
Table 16.3. The CookiePolicyOptions properties

Name	Description
`CheckConsentNeeded`	This property is assigned a function that receives an `HttpContext` object and returns `true` if it represents a request for which cookie consent is required. The function is called for every request, and the default function always returns `false`.
`ConsentCookie`	This property returns an object that is used to configure the cookie sent to the browser to record the user's cookie consent.
`HttpOnly`	This property sets the default value for the `HttpOnly` property, as described in table 16.2.
`MinimumSameSitePolicy`	This property sets the lowest level of security for the `SameSite` property, as described in table 16.2.
`Secure`	This property sets the default value for the `Secure` property, as described in table 16.2.

To enable consent checking, I assigned a new function to the `CheckConsentNeeded` property that always returns `true`. The function is called for every request that ASP

.NET Core receives, which means that sophisticated rules can be defined to select the requests for which consent is required. For this application, I have taken the most cautious approach and required consent for all requests.

The middleware that enforces the cookie policy is added to the request pipeline using the `UseCookiePolicy` method. The result is that only cookies whose `IsEssential` property is `true` will be added to responses. Listing 16.4 sets the `IsEssential` property on `cookie1` only, and you can see the effect by restarting ASP.NET Core, requesting http://localhost:5000/cookie, and reloading the browser. Only the counter whose cookie is marked as essential updates, as shown in figure 16.3.



```
Counter1: 1, Counter2: 1
```

```
Counter1: 2, Counter2: 1
```

Figure 16.3 Using cookie consent

16.2.2 Managing cookie consent

Unless the user has given consent, only cookies that are essential to the core features of the web application are allowed. Consent is managed through a *request feature*, which provides middleware components with access to the implementation details of how requests and responses are handled by ASP.NET Core. Features are accessed through the `HttpRequest.Features` property, and each feature is represented by an interface whose properties and methods deal with one aspect of low-level request handling.

Features deal with aspects of request handling that rarely need to be altered, such as the structure of responses. The exception is the management of cookie consent, which is handled through the `ITrackingConsentFeature` interface, which defines the methods and properties described in table 16.4.

Table 16.4. The ITrackingConsentFeature members

Name	Description
`CanTrack`	This property returns `true` if nonessential cookies can be added to the current request, either because the user has given consent or because consent is not required.
`CreateConsentCookie()`	This method returns a cookie that can be used by JavaScript clients to indicate consent.
`GrantConsent()`	Calling this method adds a cookie to the response that grants consent for nonessential cookies.
`HasConsent`	This property returns `true` if the user has given consent for nonessential cookies.
`IsConsentNeeded`	This property returns `true` if consent for nonessential cookies is required for the current request.
`WithdrawConsent()`	This method deletes the consent cookie.

To deal with consent, add a class file named `ConsentMiddleware.cs` to the `Platform` folder and the code shown in listing 16.5. Managing cookie consent can be done using lambda expressions, but I have used a class in this example to keep the `Program.cs` method uncluttered.

Listing 16.5 The contents of the ConsentMiddleware.cs file in the Platform folder

```
using Microsoft.AspNetCore.Http.Features;

namespace Platform {
    public class ConsentMiddleware {
        private RequestDelegate next;

        public ConsentMiddleware(RequestDelegate nextDelgate) {
            next = nextDelgate;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/consent") {
                ITrackingConsentFeature? consentFeature
                    = context.Features.Get<ITrackingConsentFeature>();
                if (consentFeature != null) {
                    if (!consentFeature.HasConsent) {
                        consentFeature.GrantConsent();
                    } else {
                        consentFeature.WithdrawConsent();
                    }
                    await context.Response.WriteAsync(
                        consentFeature.HasConsent ? "Consent Granted \n"
                            : "Consent Withdrawn\n");
                }
            } else {
                await next(context);
            }
        }
    }
}
```

Request features are obtained using the `Get` method, where the generic type argument specifies the feature interface that is required, like this:

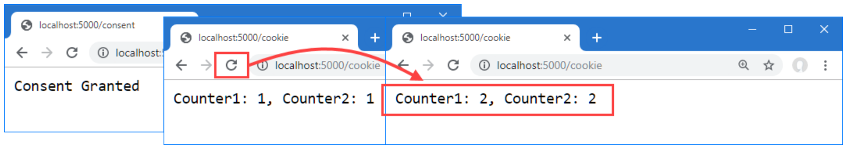
```
...
ITrackingConsentFeature? consentFeature
    = context.Features.Get<ITrackingConsentFeature>();
...
```

Using the properties and methods described in table 16.4, the new middleware component responds to the `/consent` URL to determine and change the cookie consent. Listing 16.6 adds the new middleware to the request pipeline.

Listing 16.6 Adding middleware in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<CookiePolicyOptions>(opts => {
    opts.CheckConsentNeeded = context => true;
});

var app = builder.Build();

app.UseCookiePolicy();
app.UseMiddleware<Platform.ConsentMiddleware>();

app.MapGet("/cookie", async context => {

// ...statments omitted for brevity...
```

To see the effect, restart ASP.NET Core and request http://localhost:5000/consent and then http://localhost:5000/cookie. When consent is granted, nonessential cookies are allowed, and both the counters in the example will work, as shown in figure 16.4. Repeat the process to withdraw consent, and you will find that only the counter whose cookie has been denoted as essential works.



```
Consent Granted
```

```
Counter1: 1, Counter2: 1
```

```
Counter1: 2, Counter2: 2
```

Figure 16.4 Managing cookie consent

16.3 Using sessions

The example in the previous section used cookies to store the application's state data, providing the middleware component with the data required. The problem with this approach is that the contents of the cookie are stored at the client, where it can be manipulated and used to alter the behavior of the application.

A better approach is to use the ASP.NET Core session feature. The session middleware adds a cookie to responses, which allows related requests to be identified and which is also associated with data stored at the server.

When a request containing the session cookie is received, the session middleware component retrieves the server-side data associated with the session and makes it available to other middleware components through the `HttpContext` object. Using sessions means that the application's data remains at the server and only the identifier for the session is sent to the browser.

16.3.1 Configuring the session service and middleware

Setting up sessions requires configuring services and adding a middleware component to the request pipeline. Listing 16.7 adds the statements to the `Program.cs` file to set up sessions for the example application and removes the endpoints from the previous section.

Listing 16.7 Configuring sessions in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(opts => {
    opts.IdleTimeout = TimeSpan.FromMinutes(30);
    opts.Cookie.IsEssential = true;
});

var app = builder.Build();

app.UseSession();

app.MapFallback(async context =>
    await context.Response.WriteAsync("Hello World!"));

app.Run();
```

When you use sessions, you must decide how to store the associated data. ASP.NET Core provides three options for session data storage, each of which has its own method to register a service, as described in table 16.5.

Table 16.5 The session storage methods

Name	Description
`AddDistributedMemoryCache`	This method sets up an in-memory cache. Despite the name, the cache is not distributed and is responsible only for storing data for the instance of the ASP.NET Core runtime where it is created.
`AddDistributedSqlServerCache`	This method sets up a cache that stores data in SQL Server and is available when the `Microsoft.Extensions.Caching.SqlServer` package is installed. This cache is used in chapter 17.
`AddStackExchangeRedisCache`	This method sets up a Redis cache and is available when the `Microsoft.Extensions.Caching.Redis` package is installed.

Caching is described in detail in chapter 17, but for this chapter, I used the in-memory cache:

```
...
builder.Services.AddDistributedMemoryCache();
...
```

Despite its name, the cache service created by the `AddDistributedMemoryCache` method isn't distributed and stores the session data for a single instance of the ASP.NET Core runtime. If you scale an application by deploying multiple instances of the runtime, then you should use one of the other caches, such as the SQL Server cache, which is demonstrated in chapter 17.

The next step is to use the options pattern to configure the session middleware, like this:

```
...
builder.Services.AddSession(opts => {
    opts.IdleTimeout = TimeSpan.FromMinutes(30);
    opts.Cookie.IsEssential = true;
});
...
```

Table 16.6 shows that the options class for sessions is `SessionOptions` and describes the key properties it defines.

Table 16.6 Properties defined by the SessionOptions class

Name	Description
`Cookie`	This property is used to configure the session cookie.
`IdleTimeout`	This property is used to configure the time span after which a session expires.

The `Cookie` property returns an object that can be used to configure the session cookie. Table 16.7 describes the most useful cookie configuration properties for session data.

Table 16.7 Cookie configuration properties

Name	Description
`HttpOnly`	This property specifies whether the browser will prevent the cookie from being included in HTTP requests sent by JavaScript code. This property should be set to `true` for projects that use a JavaScript application whose requests should be included in the session. The default value is `true`.
`IsEssential`	This property specifies whether the cookie is required for the application to function and should be used even when the user has specified that they don't want the application to use cookies. The default value is `false`. See the "Managing Cookie Consent" section for more details.
`SecurityPolicy`	This property sets the security policy for the cookie, using a value from the `CookieSecurePolicy` enum. The values are `Always` (which restricts the cookie to HTTPS requests), `SameAsRequest` (which restricts the cookie to HTTPS if the original request was made using HTTPS), and `None` (which allows the cookie to be used on HTTP and HTTPS requests). The default value is `None`.

The options set in listing 16.7 allow the session cookie to be included in requests started by JavaScript and flag the cookie as essential so that it will be used even when the user has expressed a preference not to use cookies (see the "Managing Cookie Consent" section for more details about essential cookies). The `IdleTimeout` option has been set so that sessions expire if no request containing the sessions cookie is received for 30 minutes.

CAUTION The session cookie isn't denoted as essential by default, which can cause problems when cookie consent is used. Listing 16.7 sets the `IsEssential` property to `true` to ensure that sessions always work. If you find sessions don't work as expected, then this is the likely cause, and you must either set `IsEssential` to `true` or adapt your application to deal with users who don't grant consent and won't accept session cookies.

The final step is to add the session middleware component to the request pipeline, which is done with the `UseSession` method. When the middleware processes a request that contains a session cookie, it retrieves the session data from the cache and makes it available through the `HttpContext` object, before passing the request along the request pipeline and providing it to other middleware components. When a request arrives without a session cookie, a new session is started, and a cookie is added to the response so that subsequent requests can be identified as being part of the session.

16.3.2 Using session data

The session middleware provides access to details of the session associated with a request through the `Session` property of the `HttpContext` object. The `Session` property returns an object that implements the `ISession` interface, which provides the methods shown in table 16.8 for accessing session data.

Table 16.8 Useful ISession methods and extension methods

Name	Description
`Clear()`	This method removes all the data in the session.
`CommitAsync()`	This asynchronous method commits changed session data to the cache.
`GetString(key)`	This method retrieves a string value using the specified key.
`GetInt32(key)`	This method retrieves an integer value using the specified key.
`Id`	This property returns the unique identifier for the session.
`IsAvailable`	This returns `true` when the session data has been loaded.
`Keys`	This enumerates the keys for the session data items.
`Remove(key)`	This method removes the value associated with the specified key.
`SetString(key,val)`	This method stores a string using the specified key.
`SetInt32(key, val)`	This method stores an integer using the specified key.

Session data is stored in key-value pairs, where the keys are strings and the values are strings or integers. This simple data structure allows session data to be stored easily by each of the caches listed in table 16.5. Applications that need to store more complex data can use serialization, which is the approach I took for the SportsStore. Listing 16.8 uses session data to re-create the counter example.

Listing 16.8 Using session data in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(opts => {
    opts.IdleTimeout = TimeSpan.FromMinutes(30);
    opts.Cookie.IsEssential = true;
});

var app = builder.Build();

app.UseSession();

app.MapGet("/session", async context => {
    int counter1 = (context.Session.GetInt32("counter1") ?? 0) + 1;
    int counter2 = (context.Session.GetInt32("counter2") ?? 0) + 1;
    context.Session.SetInt32("counter1", counter1);
    context.Session.SetInt32("counter2", counter2);
    await context.Session.CommitAsync();
    await context.Response
        .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
});

app.MapFallback(async context =>
    await context.Response.WriteAsync("Hello World!"));

app.Run();
```

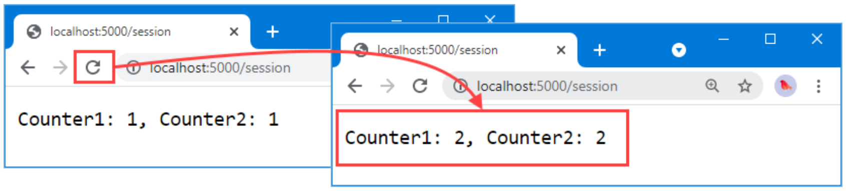
The `GetInt32` method is used to read the values associated with the keys `counter1` and `counter2`. If this is the first request in a session, no value will be available, and the null-coalescing operator is used to provide an initial value. The value is incremented and then stored using the `SetInt32` method and used to generate a simple result for the client.

The use of the `CommitAsync` method is optional, but it is good practice to use it because it will throw an exception if the session data can't be stored in the cache. By default, no error is reported if there are caching problems, which can lead to unpredictable and confusing behavior.

All changes to the session data must be made before the response is sent to the client, which is why I read, update, and store the session data before calling the `Response.WriteAsync` method in listing 16.8.

Notice that the statements in listing 16.8 do not have to deal with the session cookie, detect expired sessions, or load the session data from the cache. All this work is done automatically by the session middleware, which presents the results through the `HttpContext.Session` property. One consequence of this approach is that the `HttpContext.Session` property is not populated with data until after the session middleware has processed a request, which means that you should attempt to access session data only in middleware or endpoints that are added to the request pipeline after the `UseSession` method is called.

Restart ASP.NET Core and navigate to the http://localhost:5000/session URL, and you will see the value of the counter. Reload the browser, and the counter values will be incremented, as shown in figure 16.5. The sessions and session data will be lost when ASP.NET Core is stopped because I chose the in-memory cache. The other storage options operate outside of the ASP.NET Core runtime and survive application restarts.



```
Counter1: 1, Counter2: 1
```

```
Counter1: 2, Counter2: 2
```

Figure 16.5 Using session data

16.4 Working with HTTPS connections

Users increasingly expect web applications to use HTTPS connections, even for requests that don't contain or return sensitive data. ASP.NET Core supports both HTTP and HTTPS connections and provides middleware that can force HTTP clients to use HTTPS.

HTTPS vs. SSL vs. TLS

HTTPS is the combination of HTTP and the Transport Layer Security (TLS) or Secure Sockets Layer (SSL). TLS has replaced the obsolete SSL protocol, but the term SSL has become synonymous with secure networking and is often used to refer to TLS. If you are interested in security and cryptography, then the details of HTTPS are worth exploring, and https://en.wikipedia.org/wiki/HTTPS is a good place to start.

16.4.1 Enabling HTTPS connections

HTTPS is enabled and configured in the `launchSettings.json` file in the `Properties` folder, as shown in listing 16.9.

Listing 16.9 Changes in the launchSettings.json file in the Platform/Properties folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "Platform": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000;https://localhost:5500",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

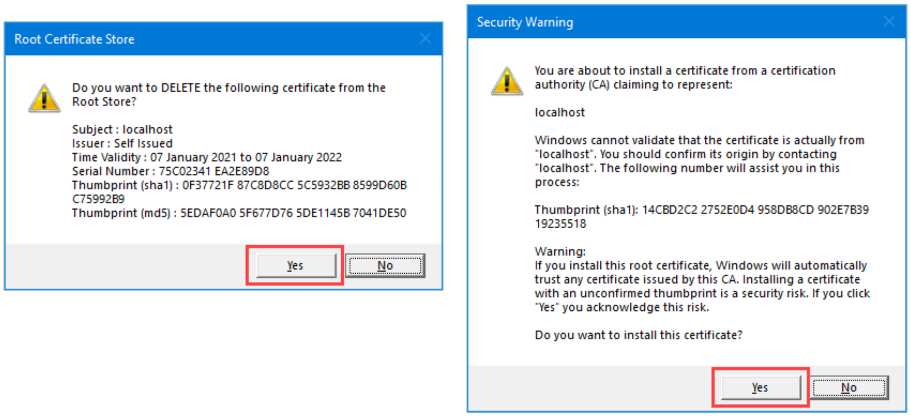
The new `applicationUrl` setting sets the URLs to which the application will respond, and HTTPS is enabled by adding an HTTPS URL to the configuration setting. Note that the URLs are separated by a semicolon and no spaces are allowed.

The .NET Core runtime includes a test certificate that is used for HTTPS requests. Run the commands shown in listing 16.10 in the `Platform` folder to regenerate and trust the test certificate.

Listing 16.10 Regenerating the Development Certificates

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Select Yes to the prompts to delete the existing certificate that has already been trusted and select Yes to trust the new certificate, as shown in figure 16.6.


Root Certificate Store

Do you want to DELETE the following certificate from the Root Store?

Subject : localhost
Issuer : Self Issued
Time Validity : 07 January 2021 to 07 January 2022
Serial Number : 75C02341 EA2E89D8
Thumbprint (sha1) : 0F37721F 87C8D8CC 5C5932BB 8599D60B C75992B9
Thumbprint (md5) : 5EDAF0A0 5F677D76 5DE1145B 7041DE50

Security Warning

You are about to install a certificate from a certification authority (CA) claiming to represent:

localhost

Windows cannot validate that the certificate is actually from "localhost". You should confirm its origin by contacting "localhost". The following number will assist you in this process:

Thumbprint (sha1): 14CBD2C2 2752E0D4 958DB8CD 902E7B39 19235518

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?

Figure 16.6 Regenerating the HTTPS certificate

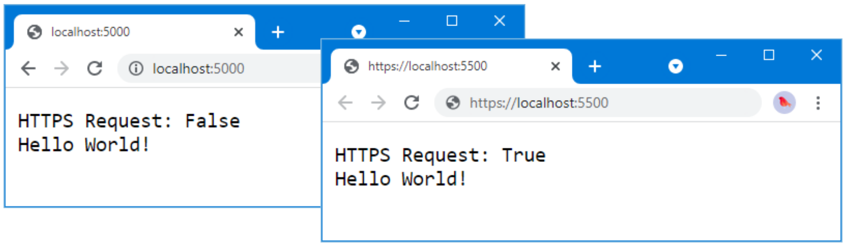
16.4.2 Detecting HTTPS requests

Requests made using HTTPS can be detected through the `HttpRequest.IsHttps` property. In listing 16.11, I added a message to the fallback response that reports whether a request is made using HTTPS.

Listing 16.11 Detecting HTTPS in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(opts => {
    opts.IdleTimeout = TimeSpan.FromMinutes(30);
    opts.Cookie.IsEssential = true;
});

var app = builder.Build();

app.UseSession();

app.MapGet("/session", async context => {
    int counter1 = (context.Session.GetInt32("counter1") ?? 0) + 1;
    int counter2 = (context.Session.GetInt32("counter2") ?? 0) + 1;
    context.Session.SetInt32("counter1", counter1);
    context.Session.SetInt32("counter2", counter2);
    await context.Session.CommitAsync();
    await context.Response
        .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
});

app.MapFallback(async context => {
    await context.Response
        .WriteAsync($"HTTPS Request: {context.Request.IsHttps} \n");
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

To test HTTPS, restart ASP.NET Core and navigate to http://localhost:5000. This is a regular HTTP request and will produce the result shown on the left of figure 16.7. Next, navigate to https://localhost:5500, paying close attention to the URL scheme, which is `https` and not `http`, as it has been in previous examples. The new middleware will detect the HTTPS connection and produce the output on the right of figure 16.7.



```
HTTPS Request: False
Hello World!
```

```
HTTPS Request: True
Hello World!
```

Figure 16.7 Detecting an HTTPS request

16.4.3 Enforcing HTTPS requests

ASP.NET Core provides a middleware component that enforces the use of HTTPS by sending a redirection response for requests that arrive over HTTP. Listing 16.12 adds this middleware to the request pipeline.

Listing 16.12 Enforcing HTTPS in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(opts => {
    opts.IdleTimeout = TimeSpan.FromMinutes(30);
    opts.Cookie.IsEssential = true;
});

var app = builder.Build();

app.UseHttpsRedirection();
app.UseSession();

app.MapGet("/session", async context => {
    int counter1 = (context.Session.GetInt32("counter1") ?? 0) + 1;
    int counter2 = (context.Session.GetInt32("counter2") ?? 0) + 1;
    context.Session.SetInt32("counter1", counter1);
    context.Session.SetInt32("counter2", counter2);
    await context.Session.CommitAsync();
    await context.Response
        .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
});

app.MapFallback(async context => {
    await context.Response
        .WriteAsync($"HTTPS Request: {context.Request.IsHttps} \n");
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

The `UseHttpsRedirection` method adds the middleware component, which appears at the start of the request pipeline so that the redirection to HTTPS occurs before any other component can short-circuit the pipeline and produce a response using regular HTTP.

Configuring HTTPS redirection

The options pattern can be used to configure the HTTPS redirection middleware, by calling the `AddHttpsRedirection` method like this:

```
...
builder.Services.AddHttpsRedirection(opts => {
```

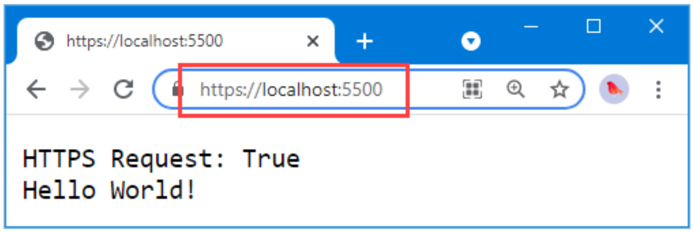
(continued)

```
    opts.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
    opts.HttpsPort = 443;
});
...
```

The only two configuration options are shown in this fragment, which sets the status code used in the redirection response, and the port to which the client is redirected, overriding the value that is loaded from the configuration files. Specifying the HTTPS port can be useful when deploying the application, but care should be taken when changing the redirection status code.

Restart ASP.NET Core and request http://localhost:5000, which is the HTTP URL for the application. The HTTPS redirection middleware will intercept the request and redirect the browser to the HTTPS URL, as shown in figure 16.8.

TIP Modern browsers often hide the URL scheme, which is why you should pay attention to the port number that is displayed. To display the URL scheme in the figure, I had to click the URL bar so the browser would display the full URL.



```
HTTPS Request: True
Hello World!
```

Figure 16.8 Forcing HTTPS requests

16.4.4 *Enabling HTTP strict transport security*

One limitation of HTTPS redirection is that the user can make an initial request using HTTP before being redirected to a secure connection, presenting a security risk.

The HTTP Strict Transport Security (HSTS) protocol is intended to help mitigate this risk and works by including a header in responses that tells browsers to use HTTPS only when sending requests to the web application's host. After an HSTS header has been received, browsers that support HSTS will send requests to the application using HTTPS even if the user specifies an HTTP URL. Listing 16.13 shows the addition of the HSTS middleware to the request pipeline.

Listing 16.13 Enabling HSTS in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(opts => {
    opts.IdleTimeout = TimeSpan.FromMinutes(30);
    opts.Cookie.IsEssential = true;
});

builder.Services.AddHsts(opts => {
    opts.MaxAge = TimeSpan.FromDays(1);
    opts.IncludeSubDomains = true;
});

var app = builder.Build();

if (app.Environment.IsProduction()) {
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseSession();

app.MapGet("/session", async context => {
    int counter1 = (context.Session.GetInt32("counter1") ?? 0) + 1;
    int counter2 = (context.Session.GetInt32("counter2") ?? 0) + 1;
    context.Session.SetInt32("counter1", counter1);
    context.Session.SetInt32("counter2", counter2);
    await context.Session.CommitAsync();
    await context.Response
        .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
});

app.MapFallback(async context => {
    await context.Response
        .WriteAsync($"HTTPS Request: {context.Request.IsHttps} \n");
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

The middleware is added to the request pipeline using the `UseHsts` method. The HSTS middleware can be configured with the `AddHsts` method, using the properties described in table 16.9.

Table 16.9 The HSTS configuration properties

Name	Description
`ExcludeHosts`	This property returns a `List<string>` that contains the hosts for which the middleware won't send an HSTS header. The defaults exclude `localhost` and the loopback addresses for IP version 4 and version 6.
`IncludeSubDomains`	When `true`, the browser will apply the HSTS setting to subdomains. The default value is `false`.
`MaxAge`	This property specifies the period for which the browser should make only HTTPS requests. The default value is 30 days.
`Preload`	This property is set to `true` for domains that are part of the HSTS preload scheme. The domains are hard-coded into the browser, which avoids the initial insecure request and ensures that only HTTPS is used. See `hstspreload.org` for more details.

HSTS is disabled during development and enabled only in production, which is why the `UseHsts` method is called only for that environment.

```
...
if (app.Environment.IsProduction()) {
    app.UseHsts();
}
...
```

HSTS must be used with care because it is easy to create a situation where clients cannot access the application, especially when nonstandard ports are used for HTTP and HTTPS.

If the example application is deployed to a server named `myhost`, for example, and the user requests http://myhost:5000, the browser will be redirected to https://myhost:5500 and sent the HSTS header, and the application will work as expected. But the next time the user requests http://myhost:5000, they will receive an error stating that a secure connection cannot be established.

This problem arises because some browsers take a simplistic approach to HSTS and assume that HTTP requests are handled on port 80 and HTTPS requests on port 443.

When the user requests http://myhost:5000, the browser checks its HSTS data and sees that it previously received an HSTS header for `myhost`. Instead of the HTTP URL that the user entered, the browser sends a request to https://myhost:5000. ASP.NET Core doesn't handle HTTPS on the port it uses for HTTP, and the request fails. The browser doesn't remember or understand the redirection it previously received for port 5001.

This isn't an issue where port 80 is used for HTTP and 443 is used for HTTPS. The URL http://myhost is equivalent to http://myhost:80, and https://myhost is equivalent to https://myhost:443, which means that changing the scheme targets the right port.

Once a browser has received an HSTS header, it will continue to honor it for the duration of the header's `MaxAge` property. When you first deploy an application, it is a good idea to set the HSTS `MaxAge` property to a relatively short duration until you are confident that your HTTPS infrastructure is working correctly, which is why I have set `MaxAge` to one day in listing 16.13. Once you are sure that clients will not need to make HTTP requests, you can increase the `MaxAge` property. A `MaxAge` value of one year is commonly used.

TIP If you are testing HSTS with Google Chrome, you can inspect and edit the list of domains to which HSTS is applied by navigating to `chrome://net-internals/#hsts`.

16.5 Using rate limits

ASP.NET Core includes middleware components that limit the rate at which requests are processed, which can be a good way to ensure that a large number of requests doesn't overwhelm the application. The .NET framework provides a general API for rate limiting, which is integrated into ASP.NET Core through extension methods used in the `Program.cs` file. Listing 16.14 defines a rate limit and applies it to an endpoint.

CAUTION Use this feature with caution. Once a rate limit has been reached, ASP.NET Core rejects HTTP requests with an error until capacity becomes available again, which can confuse clients and users alike.

Listing 16.14 Defining a rate limit in the Program.cs file in the Platform folder

```
using Microsoft.AspNetCore.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(opts => {
    opts.IdleTimeout = TimeSpan.FromMinutes(30);
    opts.Cookie.IsEssential = true;
});

builder.Services.AddHsts(opts => {
    opts.MaxAge = TimeSpan.FromDays(1);
    opts.IncludeSubDomains = true;
});

builder.Services.AddRateLimiter(opts => {
    opts.AddFixedWindowLimiter("fixedWindow", fixOpts => {
        fixOpts.PermitLimit = 1;
        fixOpts.QueueLimit = 0;
        fixOpts.Window = TimeSpan.FromSeconds(15);
    });
});

var app = builder.Build();

if (app.Environment.IsProduction()) {
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseRateLimiter();

app.UseSession();

app.MapGet("/session", async context => {
    int counter1 = (context.Session.GetInt32("counter1") ?? 0) + 1;
    int counter2 = (context.Session.GetInt32("counter2") ?? 0) + 1;
    context.Session.SetInt32("counter1", counter1);
    context.Session.SetInt32("counter2", counter2);
    await context.Session.CommitAsync();
    await context.Response
        .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
}).RequireRateLimiting("fixedWindow");
```

```
app.MapFallback(async context => {
    await context.Response
        .WriteAsync($"HTTPS Request: {context.Request.IsHttps} \n");
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

The `AddRateLimiter` extension method is used to configure rate limiting, which is done using the options pattern. In this example, I have used the `AddFixedWindowLimiter` method to create a rate-limiting policy that limits the number of requests that will be handled in a specified duration. The `AddFixedWindowLimiter` method is one of four extension methods that are available for rate limiting, described in table 16.10. Full details of how each of these rate limits works can be found at https://learn.microsoft.com/en-us/aspnet/core/performance/rate-limit.

Table 16.10 The rate limiting extension methods

Name	Description
`AddFixedWindowLimiter`	This method creates a rate limiter that allows a specified number of requests in a fixed period.
`AddSlidingWindowLimiter`	This method creates a rate limiter that allows a specified number of requests in a fixed period, with the addition of a sliding window to smooth the rate limits.
`AddTokenBucketLimiter`	This method creates a rate limiter that maintains a pool of tokens that are allocated to requests. Requests can be allocated different amounts of tokens, and requests are only handled when there are sufficient free tokens in the pool.
`AddConcurrencyLimiter`	This method creates a rate limiter that allows a specific number of concurrent requests.

This is the least flexible of the time-based rate limiters, but it is the easiest to demonstrate and test:

```
...
opts.AddFixedWindowLimiter("fixedWindow", fixOpts => {
    fixOpts.PermitLimit = 1;
    fixOpts.QueueLimit = 0;
    fixOpts.Window = TimeSpan.FromSeconds(15);
});
...
```

Each extension method is configured with an instance of its own options class, but they all share the most important properties. The `PermitLimit` property is used to specify the maximum number of requests, and the `QueueLimit` property is used to specify the maximum number of requests that will be queued waiting for available capacity. If there is no available capacity and no available slots in the queue, then requests will be rejected. The combination of properties is given a name, which is used to apply the rate limit to endpoints.

These options are supplemented by additional properties which are specific to each rate limiter. In the case of the `AddFixedWindowLimiter` method, the `Window` property is used to specify the duration to which the rate is applied.

In listing 16.14, I specified a `PermitLimit` of `1`, a `QueueLimit` of 0, and a `Window` of `15` seconds. This means that one request will be accepted every 15 seconds, without any queue, meaning that any additional requests will be rejected. This rate limit is assigned the name `fixedWindow`.

The rate limiting middleware is added to the pipeline using the `UseRateLimiter` method and applied to an endpoint with the `RequireRateLimiting` method. An application can define multiple rate limits and so a name is used to select the rate limit that is required:

```
...
}).RequireRateLimiting("fixedWindow");
...
```

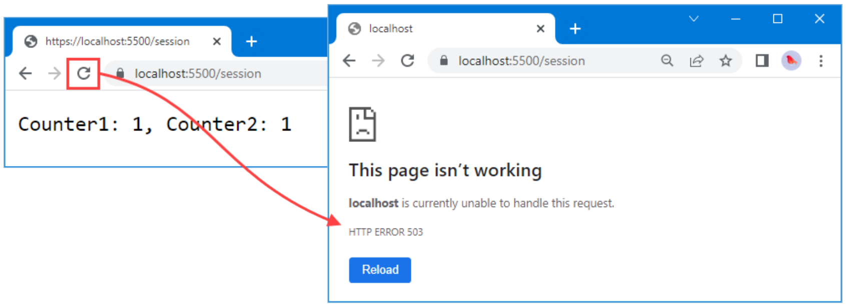
Endpoints can be con configured with different rate limits, or no rate limit, and each rate limit will be managed independently.

Applying rate limits to controllers and pages

ASP.NET Core provides extension methods to configure rate limits for controllers, described in chapter 19, and Razor Pages, which are described in chapter 23.

It can be difficult to test rate limits effectively because browsers will often apply their own restrictions on the requests they send. Microsoft provides recommendations for testing tools at https://learn.microsoft.com/en-us/aspnet/core/performance/rate-limit#testing-endpoints-with-rate-limiting, but a policy as simple as the one defined in listing 16.14 is easy to test.

Restart ASP.NET Core and request https://localhost:5500/session. Click the Reload button within 15 seconds, and the new request will exceed the rate limit and ASP.NET Core will respond with a 503 status code, as shown in figure 16.9. Wait until the 15-second period has elapsed and click the Reload button again; the rate limit should reset and the request will be processed normally.



```
Counter1: 1, Counter2: 1
```

This page isn't working

localhost is currently unable to handle this request.

HTTP ERROR 503

Figure 16.9 The effect of a rate limit.

16.6 Handling exceptions and errors

When the request pipeline is created, the `WebApplicationBuilder` class uses the development environment to enable middleware that handles exceptions by producing HTTP responses that are helpful to developers. Here is a fragment of code from the `WebApplicationBuilder` class:

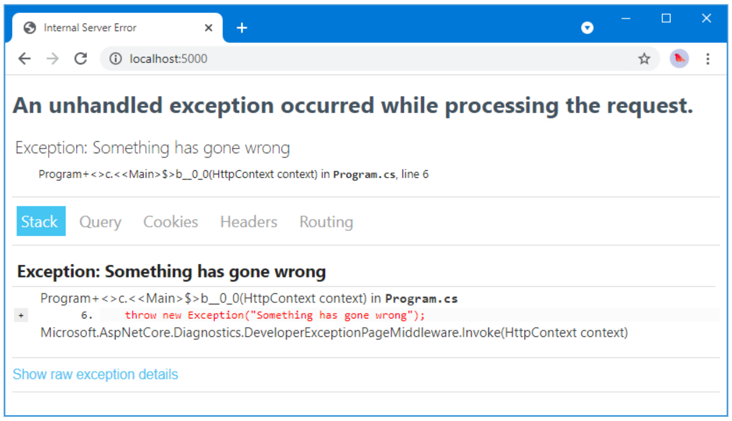
```
...
if (context.HostingEnvironment.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
}
...
```

The `UseDeveloperExceptionPage` method adds the middleware component that intercepts exceptions and presents a more useful response. To demonstrate the way that exceptions are handled, listing 16.15 replaces the middleware and endpoints used in earlier examples with a new component that deliberately throws an exception.

Listing 16.15 Adding Middleware in the Program.cs File in the Platform Folder

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.Run(context => {
    throw new Exception("Something has gone wrong");
});

app.Run();
```

Restart ASP.NET Core and navigate to http://localhost:5000 to see the response that the middleware component generates, which is shown in figure 16.10. The page presents a stack trace and details about the request, including details of the headers and cookies it contained.



An unhandled exception occurred while processing the request.

Exception: Something has gone wrong

Program+<>c.<<Main>$>b__0_0(HttpContext context) in Program.cs, line 6

Exception: Something has gone wrong

```
Program+<>c.<<Main>$>b__0_0(HttpContext context) in Program.cs
+       6.      throw new Exception("Something has gone wrong");
Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware.Invoke(HttpContext context)
```

Figure 16.10 The developer exception page

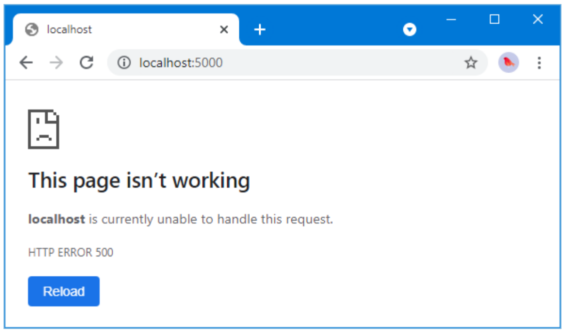
16.6.1 Returning an HTML error response

When the developer exception middleware is disabled, as it will be when the application is in production, ASP.NET Core deals with unhandled exceptions by sending a response that contains just an error code. Listing 16.16 changes the environment to production.

Listing 16.16 Changes in the launchSettings.json file in the Platform/Properties folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "Platform": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000;https://localhost:5500",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Start ASP.NET Core using the `dotnet run` command and navigate to http://localhost:5000. The response you see will depend on your browser because ASP.NET Core has only provided it with a response containing status code 500, without any content to display. Figure 16.11 shows how this is handled by Google Chrome.



This page isn't working

localhost is currently unable to handle this request.

HTTP ERROR 500

Figure 16.11 Returning an error response

As an alternative to returning just status codes, ASP.NET Core provides middleware that intercepts unhandled exceptions and sends a redirection to the browser instead, which can be used to show a friendlier response than the raw status code. The exception redirection middleware is added with the `UseExceptionHandler` method, as shown in listing 16.17.

Listing 16.17 Returning an error response in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

if (!app.Environment.IsDevelopment()) {
    app.UseExceptionHandler("/error.html");
    app.UseStaticFiles();
}

app.Run(context => {
    throw new Exception("Something has gone wrong");
});

app.Run();
```

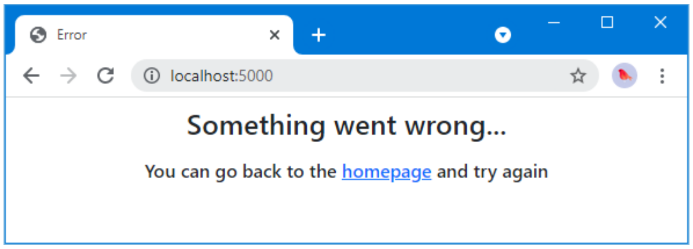
When an exception is thrown, the exception handler middleware will intercept the response and redirect the browser to the URL provided as the argument to the `UseExceptionHandler` method. For this example, the redirection is to a URL that will be handled by a static file, so the `UseStaticFiles` middleware has also been added to the pipeline.

To add the file that the browser will receive, create an HTML file named `error.html` in the `wwwroot` folder and add the content shown in listing 16.18.

Listing 16.18 The contents of the error.html file in the Platform/wwwroot folder

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="/lib/bootstrap/css/bootstrap.min.css" />
    <title>Error</title>
</head>
<body class="text-center">
    <h3 class="p-2">Something went wrong...</h3>
    <h6>You can go back to the <a href="/">homepage</a> and try again</h6>
</body>
</html>
```

Restart ASP.NET Core and navigate to http://localhost:5000 to see the effect of the new middleware. Instead of the raw status code, the browser will be sent the content of the `/error.html` URL, as shown in figure 16.12.



Something went wrong...

You can go back to the homepage and try again

Figure 16.12 Displaying an HTML error

There are versions of the `UseExceptionHandler` method that allow more complex responses to be composed, but my advice is to keep error handling as simple as possible because you can't anticipate all of the problems an application may encounter, and you run the risk of encountering another exception when trying to handle the one that triggered the handler, resulting in a confusing response or no response at all.

16.6.2 Enriching status code responses

Not all error responses will be the result of uncaught exceptions. Some requests cannot be processed for reasons other than software defects, such as requests for URLs that are not supported or that require authentication. For this type of problem, redirecting the client to a different URL can be problematic because some clients rely on the error code to detect problems. You will see examples of this in later chapters when I show you how to create and consume RESTful web applications.

ASP.NET Core provides middleware that adds user-friendly content to error responses without requiring redirection. This preserves the error status code while providing a human-readable message that helps users make sense of the problem.

The simplest approach is to define a string that will be used as the body for the response. This is more awkward than simply pointing at a file, but it is a more reliable technique, and as a rule, simple and reliable techniques are preferable when handling errors. To create the string response for the example project, add a class file named `ResponseStrings.cs` to the `Platform` folder with the code shown in listing 16.19.

Listing 16.19 The contents of the ResponseStrings.cs file in the Platform folder

```
namespace Platform {

    public static class Responses {

        public static string DefaultResponse = @"
        <!DOCTYPE html>
            <html lang=""en"">
            <head>
                <link rel=""stylesheet""
                   href=""/lib/bootstrap/css/bootstrap.min.css"" />
                <title>Error</title>
            </head>
            <body class=""text-center"">
                <h3 class=""p-2"">Error {0}</h3>
                <h6>
```

```
                    You can go back to the <a href=""/"">homepage</a>
                        and try again
                </h6>
            </body>
        </html>";
    }
}
```

The `Responses` class defines a `DefaultResponse` property to which I have assigned a multiline string containing a simple HTML document. There is a placeholder—`{0}`—into which the response status code will be inserted when the response is sent to the client.

Listing 16.20 adds the status code middleware to the request pipeline and adds a new middleware component that will return a 404 status code, indicating that the requested URL was not found.

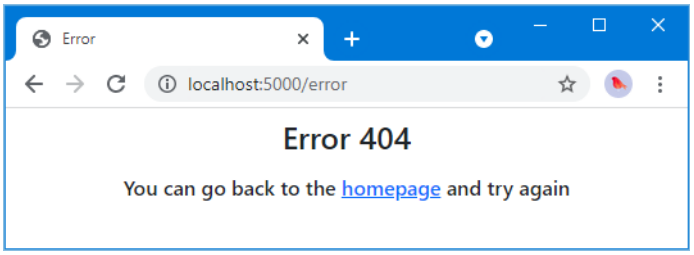
Listing 16.20 Adding middleware in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

if (!app.Environment.IsDevelopment()) {
    app.UseExceptionHandler("/error.html");
    app.UseStaticFiles();
}

app.UseStatusCodePages("text/html", Platform.Responses.DefaultResponse);

app.Use(async (context, next) => {
    if (context.Request.Path == "/error") {
        context.Response.StatusCode = StatusCodes.Status404NotFound;
        await Task.CompletedTask;
    } else {
        await next();
    }
});

app.Run(context => {
    throw new Exception("Something has gone wrong");
});

app.Run();
```

The `UseStatusCodePages` method adds the response-enriching middleware to the request pipeline. The first argument is the value that will be used for the response's `Content-Type` header, which is `text/html` in this example. The second argument is the string that will be used as the body of the response, which is the HTML string from listing 16.19.

The custom middleware component sets the `HttpResponse.StatusCode` property to specify the status code for the response, using a value defined by the `StatusCode` class. Middleware components are required to return a `Task`, so I have used the `Task.CompletedTask` property because there is no work for this middleware component to do.

To see how the 404 status code is handled, restart ASP.NET Core and request http://localhost:5000/error. The status code middleware will intercept the result and add the content shown in figure 16.13 to the response. The string used as the second argument to `UseStatusCodePages` is interpolated using the status code to resolve the placeholder.



Error 404

You can go back to the homepage and try again

Figure 16.13 Using the status code middleware

The status code middleware responds only to status codes between 400 and 600 and doesn't alter responses that already contain content, which means you won't see the response in the figure if an error occurs after another middleware component has started to generate a response. The status code middleware won't respond to unhandled exceptions because exceptions disrupt the flow of a request through the pipeline, meaning that the status code middleware isn't given the opportunity to inspect the response before it is sent to the client. As a result, the `UseStatusCodePages` method is typically used in conjunction with the `UseExceptionHandler` or `UseDeveloperExceptionPage` method.

NOTE There are two related methods, `UseStatusCodePagesWithRedirects` and `UseStatusCodePagesWithReExecute`, which work by redirecting the client to a different URL or by rerunning the request through the pipeline with a different URL. In both cases, the original status code may be lost.

16.7 *Filtering requests using the host header*

The HTTP specification requires requests to include a `Host` header that specifies the hostname the request is intended for, which makes it possible to support virtual servers where one HTTP server receives requests on a single port and handles them differently based on the hostname that was requested.

The default set of middleware that is added to the request pipeline by the `Program` class includes middleware that filters requests based on the `Host` header so that only requests that target a list of approved hostnames are handled and all other requests are rejected.

The default configuration for the `Hosts` header middleware is included in the `appsettings.json` file, as follows:

```
...
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Location": {
    "CityName": "Buffalo"
  }
}
...
```

The `AllowedHosts` configuration property is added to the JSON file when the project is created, and the default value accepts requests regardless of the `Host` header value. You can change the configuration by editing the JSON file. The configuration can also be changed using the options pattern, as shown in listing 16.21.

NOTE The middleware is added to the pipeline by default, but you can use the `UseHostFiltering` method if you need to add the middleware explicitly.

Listing 16.21 Configuring host filtering in the Program.cs file in the Platform folder

```
using Microsoft.AspNetCore.HostFiltering;

var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<HostFilteringOptions>(opts => {
    opts.AllowedHosts.Clear();
    opts.AllowedHosts.Add("*.example.com");
});

var app = builder.Build();

if (!app.Environment.IsDevelopment()) {
    app.UseExceptionHandler("/error.html");
    app.UseStaticFiles();
}

app.UseStatusCodePages("text/html", Platform.Responses.DefaultResponse);

app.Use(async (context, next) => {
    if (context.Request.Path == "/error") {
        context.Response.StatusCode = StatusCodes.Status404NotFound;
        await Task.CompletedTask;
    } else {
        await next();
    }
});
```

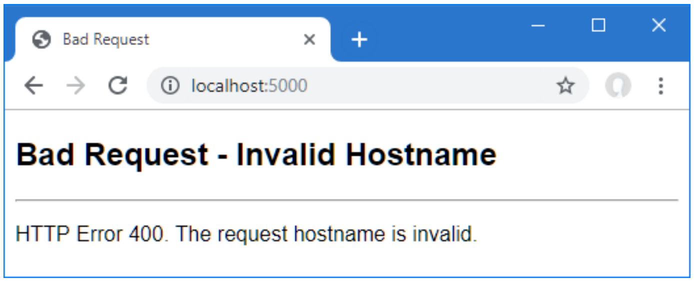
```
app.Run(context => {
    throw new Exception("Something has gone wrong");
});

app.Run();
```

The HostFilteringOptions class is used to configure the host filtering middleware using the properties described in table 16.11.

Table 16.11 The HostFilteringOptions properties

Name	Description
AllowedHosts	This property returns a `List<string>` that contains the domains for which requests are allowed. Wildcards are allowed so that `*.example.com` accepts all names in the `example.com` domain and `*` accepts all header values.
AllowEmptyHosts	When `false`, this property tells the middleware to reject requests that do not contain a `Host` header. The default value is `true`.
IncludeFailureMessage	When `true`, this property includes a message in the response that indicates the reason for the error. The default value is `true`.

In listing 16.21, I called the Clear method to remove the wildcard entry that has been loaded from the appsettings.json file and then called the Add method to accept all hosts in the example.com domain. Requests sent from the browser to localhost will no longer contain an acceptable Host header. You can see what happens by restarting ASP.NET Core and using the browser to request http://localhost:5000. The Host header middleware checks the Host header in the request, determines that the request hostname doesn't match the AllowedHosts list, and terminates the request with the 400 status code, which indicates a bad request. Figure 16.14 shows the error message.



Bad Request - Invalid Hostname

HTTP Error 400. The request hostname is invalid.

Figure 16.14 A request rejected based on the Host header

Summary

- ASP.NET Core provides support for adding cookies to responses and reading those cookies when the client includes them in subsequent requests.
- Sessions allow related requests to be identified to provide continuity across requests.
- ASP.NET Core supports HTTPS requests and can be configured to disallow regular HTTP requests.
- Limits can be applied to the rate of requests handled by endpoints.